TypeScript : POO

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com

Plan

TypeScript

Qu'est ce qu'une classe en POO ?

- $\equiv$ un plan, un moule, une usine...
- Description abstraite d'un type d'objets
- Représentant un ensemble d'objets ayant les mêmes propriétés statiques (attributs) et dynamiques (méthodes)

TypeScript

Qu'est ce qu'une classe en POO ?

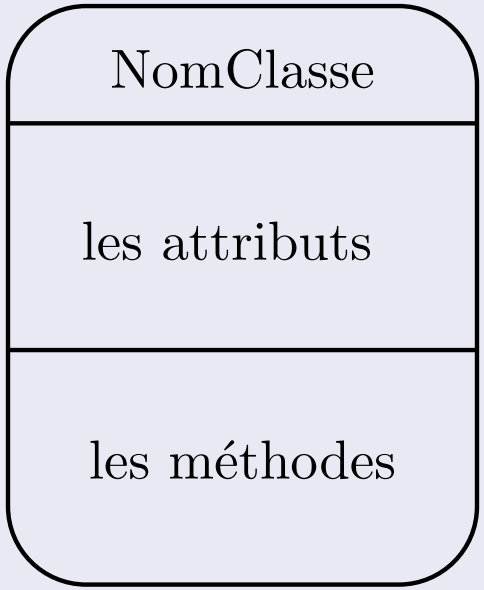
- $\equiv$ un plan, un moule, une usine...
- Description abstraite d'un type d'objets
- Représentant un ensemble d'objets ayant les mêmes propriétés statiques (attributs) et dynamiques (méthodes)

Instance ?

- Une instance correspond à un objet créé à partir d'une classe (via le constructeur)
- L'instanciation : création d'un objet d'une classe
- instance $\equiv$ objet

TypeScript

De quoi est composé une classe ?



- Attribut : [visibilité] + nom + type
- Méthode : [visibilité] + nom + arguments + valeur de retour ≡ signature : exactement comme les fonctions en procédurale

TypeScript

Considérons la classe `Personne` définie dans `personne.ts`

```
export class Personne {
    num: number;
    nom: string;
    prenom: string;
}
```

TypeScript

Considérons la classe `Personne` définie dans `personne.ts`

```
export class Personne {
    num: number;
    nom: string;
    prenom: string;
}
```


Avant de compiler, vérifiez dans `tsconfig.json` les propriétés suivantes

- `"target": "es6"`
- `"strictPropertyInitialization": false`

TypeScript

En **TypeScript**

- Toute classe a un constructeur par défaut sans paramètre.
- Par défaut, la visibilité est `public`.

TypeScript

Hypothèse

Si on voulait créer un objet de la classe `Personne` avec les valeurs `1`, `wick` et `john`

TypeScript

Hypothèse

Si on voulait créer un objet de la classe `Personne` avec les valeurs `1`, `wick` et `john`

Étape 1 : Commençons par importer la classe `Personne` dans `file.ts`

```
import { Personne } from './personne';
```

TypeScript

Hypothèse

Si on voulait créer un objet de la classe `Personne` avec les valeurs `1`, `wick` et `john`

Étape 1 : Commençons par importer la classe `Personne` dans `file.ts`

```
import { Personne } from './personne';
```

Étape 2 : déclarons un objet (objet non créé)

```
let  personne: Personne;
```

TypeScript

Hypothèse

Si on voulait créer un objet de la classe `Personne` avec les valeurs `1`, `wick` et `john`

Étape 1 : Commençons par importer la classe `Personne` dans `file.ts`

```
import { Personne } from './personne';
```

Étape 2 : déclarons un objet (objet non créé)

```
let  personne: Personne;
```

Étape 3 : créons l'objet (instanciation) de type `Personne` (objet créé)

```
personne = new Personne();
```

TypeScript

Hypothèse

Si on voulait créer un objet de la classe `Personne` avec les valeurs `1`, `wick` et `john`

Étape 1 : Commençons par importer la classe `Personne` dans `file.ts`

```
import { Personne } from './personne';
```

Étape 2 : déclarons un objet (objet non créé)

```
let  personne: Personne;
```

Étape 3 : créons l'objet (instanciation) de type `Personne` (objet créé)

```
personne = new Personne();
```

On peut faire déclaration + instanciation

```
let personne: Personne = new Personne();
```

TypeScript

Affectons les valeurs aux différents attributs

```
personne.num = 1;
personne.nom = "wick";
personne.prenom = "john";
```

TypeScript

Affectons les valeurs aux différents attributs

```
personne.num = 1;
personne.nom = "wick";
personne.prenom = "john";
```

Pour être sûr que les valeurs ont bien été affectées aux attributs, on affiche

```
console.log(personne);
// affiche Personne { num: 1, nom: 'wick', prenom: '
   john' }
```

TypeScript

Hypothèse

Supposant que l'on n'accepte pas de valeur négative pour l'attribut `num` de la classe `Personne`

TypeScript

Hypothèse

Supposant que l'on n'accepte pas de valeur négative pour l'attribut `num` de la classe `Personne`

Démarche

1. Bloquer l'accès directe aux attributs (mettre la visibilité à `private`)
2. Définir des méthodes publiques qui contrôlent l'affectation de valeurs aux attributs (les `setter`)

TypeScript

Hypothèse

Supposant que l'on n'accepte pas de valeur négative pour l'attribut `num` de la classe `Personne`

Démarche

1. Bloquer l'accès directe aux attributs (mettre la visibilité à `private`)
2. Définir des méthodes publiques qui contrôlent l'affectation de valeurs aux attributs (les `setter`)

Convention

- Mettre la visibilité `private` ou `protected` pour tous les attributs
- Mettre la visibilité `public` pour toutes les méthodes

TypeScript

Mettons la visibilité `private` pour tous les attributs de la classe `Personne`

```
export  class Personne {
    private num: number;
    private nom: string;
    private prenom: string;
}
```

TypeScript

Mettons la visibilité `private` pour tous les attributs de la classe `Personne`

```
export  class Personne {
    private num: number;
    private nom: string;
    private prenom: string;
}
```


Dans le fichier `file.ts`, les trois lignes suivantes sont soulignées en rouge

```
personne.num = 1;
personne.nom = "wick";
personne.prenom = "john";
```

TypeScript

Explication

Les attributs sont privés, donc aucun accès direct n'est autorisé.

TypeScript

Explication

Les attributs sont privés, donc aucun accès direct n'est autorisé.

Solution : les setters

Des méthodes qui contrôlent l'affectation de valeurs aux attributs.

TypeScript

Conventions **TypeScript**

- Le setter est une méthode déclarée avec le mot-clé `set`.
- Il porte le nom de l'attribut.
- On l'utilise comme un attribut.
- Pour éviter l'ambiguïté, on préfixe l'attribut par un underscore.

TypeScript

Nouveau contenu de la classe `Personne` après ajout des setters

```
export  class Personne {
    private _num: number;
    private _nom: string;
    private _prenom: string;

    public set num(_num : number) {
        this._num  = (_num >= 0 ?  _num : 0);
    }
    public set nom(_nom: string) {
        this._nom = _nom;
    }
    public set prenom(_prenom: string) {
        this._prenom = _prenom;
    }
}
```

TypeScript

Pour tester, rien à changer dans `file.ts`

```
import { Personne } from './personne';

let personne: Personne = new Personne();
personne.num = 1;
personne.nom = "wick";
personne.prenom = "john";

console.log(personne);
```

TypeScript

Pour tester, rien à changer dans `file.ts`

```
import { Personne } from './personne';

let personne: Personne = new Personne();
personne.num = 1;
personne.nom = "wick";
personne.prenom = "john";

console.log(personne);
```


Le résultat est :

```
Personne { _num: 1, _nom: 'wick', _prenom: 'john' }
```

TypeScript

Testons avec une valeur négative pour l'attribut `numero`

```
import { Personne } from './personne';

let personne: Personne = new Personne();
personne.num = -1;
personne.nom = "wick";
personne.prenom = "john";

console.log(personne);
```

TypeScript

Testons avec une valeur négative pour l'attribut `numero`

```
import { Personne } from './personne';

let personne: Personne = new Personne();
personne.num = -1;
personne.nom = "wick";
personne.prenom = "john";

console.log(personne);
```


Le résultat est :

```
Personne { _num: 0, _nom: 'wick', _prenom: 'john' }
```

TypeScript

Question

Comment récupérer les attributs (privés) de la classe `Personne` ?

TypeScript

Question

Comment récupérer les attributs (privés) de la classe `Personne` ?

Démarche

Définir des méthodes qui retournent les valeurs des attributs (les `getter`)

TypeScript

Question

Comment récupérer les attributs (privés) de la classe `Personne` ?

Démarche

Définir des méthodes qui retournent les valeurs des attributs (les `getter`)

Conventions **TypeScript**

- Le getter est une méthode déclarée avec le mot-clé `get`.
- Il porte le nom de l'attribut.
- On l'utilise comme un attribut.

TypeScript

Ajoutons les getters dans la classe `Personne`

```
public get num() : number {
    return this._num;
}
public get nom(): string {
    return this._nom;
}
public get prenom(): string {
    return this._prenom;
}
```

TypeScript

Pour tester

```
import { Personne } from './personne';

let personne: Personne = new Personne();
personne.num = 1;
personne.nom = "wick";
personne.prenom = "john";

console.log(personne.num);
// affiche 1

console.log(personne.nom);
// affiche wick

console.log(personne.prenom);
// affiche john
```

TypeScript

Depuis TypeScript 4.9, on peut utiliser le mot-clé `accessor` pour fusionner la déclaration de l'attribut et les getter et setter

```
export  class Personne {

    private  _num: number;
    accessor  nom: string;
    accessor  prenom: string;

    public set num(_num : number) {
        this._num  = (_num >= 0 ?  _num : 0);
    }
    public get num() : number {
        return this._num;
    }
}
```

TypeScript

Pour tester, on ne change rien dans `file.ts`

```
import { Personne } from './personne';

let personne: Personne = new Personne();
personne.num = 1;
personne.nom = "wick";
personne.prenom = "john";

console.log(personne.num);
// affiche 1

console.log(personne.nom);
// affiche wick

console.log(personne.prenom);
// affiche john
```

TypeScript

Remarques

- Par défaut, toute classe **TypeScript** a un constructeur par défaut sans paramètre.
- Pour simplifier la création d'objets, on peut définir un nouveau constructeur qui prend en paramètre plusieurs attributs de la classe.

TypeScript

Remarques

- Par défaut, toute classe **TypeScript** a un constructeur par défaut sans paramètre.
- Pour simplifier la création d'objets, on peut définir un nouveau constructeur qui prend en paramètre plusieurs attributs de la classe.

Les constructeurs avec **TypeScript**

- On le déclare avec le mot-clé `constructor`.
- Il peut contenir la visibilité des attributs si on veut simplifier la déclaration.

TypeScript

Le constructeur de la classe `Personne` prenant trois paramètres

```
public constructor(_num: number, _nom: string, _prenom: string)
    {
  this._num = _num;
  this._nom = _nom;
  this._prenom = _prenom;
}
```

TypeScript

Le constructeur de la classe `Personne` prenant trois paramètres

```
public constructor(_num: number, _nom: string, _prenom: string)
    {
  this._num = _num;
  this._nom = _nom;
  this._prenom = _prenom;
}
```

Pour préserver la cohérence, il faut que le constructeur contrôle la valeur de l'attribut `num`

```
public constructor(_num: number, _nom: string, _prenom: string)
    {
  this._num  = (_num >= 0 ?  _num : 0);
  this._nom = _nom;
  this._prenom = _prenom;
}
```

TypeScript

On peut aussi appelé le `setter` dans le constructeur

```
public constructor(_num: number, _nom: string,
   _prenom: string) {
  this.num = _num;
  this._nom = _nom;
  this._prenom = _prenom;
}
```

TypeScript

Dans `file.ts`, la ligne suivante est soulignée en rouge

```
let personne: Personne = new Personne();
```

TypeScript

Dans `file.ts`, la ligne suivante est soulignée en rouge

```
let personne: Personne = new Personne();
```

Explication

Le constructeur par défaut a été écrasé (il n'existe plus)

TypeScript

Dans `file.ts`, la ligne suivante est soulignée en rouge

```
let personne: Personne = new Personne();
```


Explication

Le constructeur par défaut a été écrasé (il n'existe plus)

Comment faire ?

- TypeScript n'autorise pas la présence de plusieurs constructeurs (la surcharge)
- On peut utiliser soit les valeurs par défaut, soit les paramètres optionnels

TypeScript

Le nouveau constructeur avec les paramètres optionnels

```
public constructor(_num?: number, _nom?: string,
   _prenom?: string) {
  if(_num)
    this.num = _num;
  if (_nom)
    this._nom = _nom;
  if(_prenom)
    this._prenom = _prenom;
}
```

TypeScript

Pour tester

```
import { Personne } from './personne';

let personne: Personne = new Personne();
personne.num = -1;
personne.nom = "wick";
personne.prenom = "john";
console.log(personne);

let personne2: Personne = new Personne(2, 'bob', 'mike');
console.log(personne2);
```

TypeScript

Pour tester

```
import { Personne } from './personne';

let personne: Personne = new Personne();
personne.num = -1;
personne.nom = "wick";
personne.prenom = "john";
console.log(personne);

let personne2: Personne = new Personne(2, 'bob', 'mike');
console.log(personne2);
```


En exécutant, le résultat est :

```
Personne { _num: 0, _nom: 'wick', _prenom: 'john' }
Personne { _num: 2, _nom: 'bob', _prenom: 'mike' }
```

TypeScript

TypeScript nous offre la possibilité de fusionner la déclaration des attributs et le constructeur

```
public constructor(private _num?: number,
                   private _nom?: string,
                   private _prenom?: string) {

}
```

TypeScript

TypeScript nous offre la possibilité de fusionner la déclaration des attributs et le constructeur

```
public constructor(private _num?: number,
                   private _nom?: string,
                   private _prenom?: string) {

}
```


En exécutant, le résultat est le même

```
Personne { _num: 0, _nom: 'wick', _prenom: 'john' }
Personne { _num: 2, _nom: 'bob', _prenom: 'mike' }
```

TypeScript

Récapitulatif

Les instances d'une même classe ont toutes les mêmes attributs mais pas les mêmes valeurs

TypeScript

Récapitulatif

Les instances d'une même classe ont toutes les mêmes attributs mais pas les mêmes valeurs

Hypothèse

Et si nous voulions qu'un attribut ait une valeur partagée par toutes les instances (par exemple, le nombre d'objets instanciés de la classe `Personne`)

TypeScript

Récapitulatif

Les instances d'une même classe ont toutes les mêmes attributs mais pas les mêmes valeurs

Hypothèse

Et si nous voulions qu'un attribut ait une valeur partagée par toutes les instances (par exemple, le nombre d'objets instanciés de la classe `Personne`)

Solution : attribut statique ou attribut de classe

Un attribut dont la valeur est partagée par toutes les instances de la classe.

TypeScript

Exemple

- Si on voulait créer un attribut contenant le nombre d'objets créés à partir de la classe `Personne`
- Notre attribut doit être déclaré `static`, sinon chaque objet pourrait avoir sa propre valeur pour cet attribut

TypeScript

Ajoutons un attribut statique `_nbrPersonnes` à la liste d'attributs de la classe `Personne`

```
private static _nbrPersonnes: number = 0;
```

TypeScript

Ajoutons un attribut statique `_nbrPersonnes` à la liste d'attributs de la classe `Personne`

```
private static _nbrPersonnes: number = 0;
```

Incrémentons notre compteur de personnes dans les constructeurs

```
public constructor(private _num?: number,
                   private _nom?: string,
                   private _prenom?: string) {
  Personne._nbrPersonnes++;
}
```

TypeScript

Créons un `getter` pour l'attribut static `_nbrPersonnes`

```
public static get  nbrPersonnes() {
  return Personne._nbrPersonnes;
}
```

TypeScript

Créons un `getter` pour l'attribut static `_nbrPersonnes`

```
public static get  nbrPersonnes() {
  return Personne._nbrPersonnes;
}
```

Testons cela dans `file.ts`

```
import { Personne } from './personne';

console.log(Personne.nbrPersonnes);
// affiche 0
let personne: Personne = new Personne();
personne.num = -1;
personne.nom = "wick";
personne.prenom = "john";
console.log(Personne.nbrPersonnes);
// affiche 1
let personne2: Personne = new Personne(2, 'bob', 'mike');
console.log(Personne.nbrPersonnes);
// affiche 2
```

TypeScript

Exercice

- Définir une classe `Adresse` avec trois attributs privés `rue`, `codePostal` et `ville` de type chaîne de caractère
- Définir un constructeur avec trois paramètres, les getters et setters
- Dans la classe `Personne`, ajouter un attribut `adresse` (de type `Adresse`) et définir un nouveau constructeur à quatre paramètres et le getter et le setter de ce nouvel attribut
- Dans `file.ts`, créer deux objets : un objet `adresse` (de type `Adresse`) et `personne` (de type `Personne`) prenant comme adresse l'objet `adresse`
- Afficher tous les attributs de l'objet `personne`

TypeScript

L'héritage, quand ?

- Lorsque deux ou plusieurs classes partagent plusieurs attributs (et méthodes)
- Lorsqu'une `Classe1` est (**une sorte de**) `Classe2`

TypeScript

L'héritage, quand ?

- Lorsque deux ou plusieurs classes partagent plusieurs attributs (et méthodes)
- Lorsqu'une `Classe1` est (**une sorte de**) `Classe2`

Forme générale

```
class ClasseFille extends ClasseMère
{
    // code
};
```

TypeScript

Exemple

- Un enseignant a un numéro, un nom, un prénom et un salaire

TypeScript

Exemple

- Un enseignant a un numéro, un nom, un prénom et un salaire
- Un étudiant a aussi un numéro, un nom, un prénom et un niveau

TypeScript

Exemple

- Un enseignant a un numéro, un nom, un prénom et un salaire
- Un étudiant a aussi un numéro, un nom, un prénom et un niveau
- Sémantiquement, enseignant et étudiant sont une sorte de personne

TypeScript

Exemple

- Un enseignant a un numéro, un nom, un prénom et un salaire
- Un étudiant a aussi un numéro, un nom, un prénom et un niveau
- Sémantiquement, enseignant et étudiant sont une sorte de personne
- En plus, les deux partagent plusieurs attributs tels que `numéro`, `nom` et `prénom`

TypeScript

Exemple

- Un enseignant a un numéro, un nom, un prénom et un salaire
- Un étudiant a aussi un numéro, un nom, un prénom et un niveau
- Sémantiquement, enseignant et étudiant sont une sorte de personne
- En plus, les deux partagent plusieurs attributs tels que `numéro`, `nom` et `prénom`
- Donc, on peut peut utiliser la classe `Personne` puisqu'elle contient tous les attributs `numéro`, `nom` et `prénom`

TypeScript

Exemple

- Un enseignant a un numéro, un nom, un prénom et un salaire
- Un étudiant a aussi un numéro, un nom, un prénom et un niveau
- Sémantiquement, enseignant et étudiant sont une sorte de personne
- En plus, les deux partagent plusieurs attributs tels que `numéro`, `nom` et `prénom`
- Donc, on peut peut utiliser la classe `Personne` puisqu'elle contient tous les attributs `numéro`, `nom` et `prénom`
- Les classes `Étudiant` et `Enseignant` hériteront donc (`extends`) de la classe `Personne`

TypeScript

Particularité du langage **TypeScript**

- Une classe ne peut hériter que d'une seule classe
- L'héritage multiple est donc non-autorisé.

TypeScript

Préparons la classe `Enseignant`

```
import { Personne } from "./personne";

export class Enseignant extends Personne {

}
```

TypeScript

Préparons la classe `Enseignant`

```
import { Personne } from "./personne";

export class Enseignant extends Personne {

}
```

Préparons la classe `Etudiant`

```
import { Personne } from "./personne";

export class Etudiant extends Personne {

}
```

`extends` est le mot-clé à utiliser pour définir une relation d'héritage entre deux classes

TypeScript

Ensuite

- Créer un attribut `niveau` dans la classe `Etudiant` ainsi que ses getter et setter
- Créer un attribut `salaire` dans la classe `Enseignant` ainsi que ses getter et setter

TypeScript

Pour créer un objet de type `Enseignant`

```
import { Enseignant } from './enseignant';

let enseignant: Enseignant  = new Enseignant();
enseignant.num = 3;
enseignant.nom = "green";
enseignant.prenom = "jonas";
enseignant.salaire = 1700;
console.log(enseignant);
```

TypeScript

Pour créer un objet de type `Enseignant`

```
import { Enseignant } from './enseignant';

let enseignant: Enseignant  = new Enseignant();
enseignant.num = 3;
enseignant.nom = "green";
enseignant.prenom = "jonas";
enseignant.salaire = 1700;
console.log(enseignant);
```


En exécutant, le résultat est :

```
Enseignant { _num: 3, _nom: 'green', _prenom: 'jonas', _salaire
   : 1700 }
```

TypeScript autorise la redéfinition : on peut définir un constructeur, même s'il existe dans la classe mère, qui prend plusieurs paramètres et qui utilise le constructeur de la classe mère

```
constructor(_num?: number,
            _nom?: string,
            _prenom?: string,
             private _salaire?: number) {

    super(_num, _nom, _prenom);
}
```

TypeScript autorise la redéfinition : on peut définir un constructeur, même s'il existe dans la classe mère, qui prend plusieurs paramètres et qui utilise le constructeur de la classe mère

```
constructor(_num?: number,
            _nom?: string,
            _prenom?: string,
             private _salaire?: number) {

    super(_num, _nom, _prenom);
}
```

`super()` fait appel au constructeur de la classe mère

TypeScript autorise la redéfinition : on peut définir un constructeur, même s'il existe dans la classe mère, qui prend plusieurs paramètres et qui utilise le constructeur de la classe mère

```
constructor(_num?: number,
            _nom?: string,
            _prenom?: string,
             private _salaire?: number) {

    super(_num, _nom, _prenom);
}
```

`super()` fait appel au constructeur de la classe mère

Maintenant, on peut créer un enseignant ainsi

```
let enseignant: Enseignant = new Enseignant(3, "green", "jonas"
   , 1700);
```

TypeScript autorise la redéfinition : on peut définir un constructeur, même s'il existe dans la classe mère, qui prend plusieurs paramètres et qui utilise le constructeur de la classe mère

```
constructor(_num?: number,
            _nom?: string,
            _prenom?: string,
             private _salaire?: number) {

    super(_num, _nom, _prenom);
}
```

`super()` fait appel au constructeur de la classe mère

Maintenant, on peut créer un enseignant ainsi

```
let enseignant: Enseignant = new Enseignant(3, "green", "jonas"
   , 1700);
```

Refaire la même chose pour `Etudiant`

TypeScript

À partir de la classe `Enseignant`

- On ne peut avoir accès direct à un attribut de la classe mère
- C'est-à-dire, on ne peut faire `this._num` car les attributs ont une visibilité `private`
- Pour modifier la valeur d'un attribut privé de la classe mère, il faut
  - soit utiliser les getters/setters
  - soit mettre la visibilité des attributs de la classe mère à `protected`

TypeScript

On peut créer un objet de la classe `Enseignant` ainsi

```
let enseignant: Enseignant = new Enseignant(3, "
   green", "jonas", 1700);
```

TypeScript

On peut créer un objet de la classe `Enseignant` ainsi

```
let enseignant: Enseignant = new Enseignant(3, "
   green", "jonas", 1700);
```

Ou ainsi

```
let enseignant: Personne = new Enseignant(3, "green"
   , "jonas", 1700);
```

TypeScript

On peut créer un objet de la classe `Enseignant` ainsi

```
let enseignant: Enseignant = new Enseignant(3, "
   green", "jonas", 1700);
```

Ou ainsi

```
let enseignant: Personne = new Enseignant(3, "green"
   , "jonas", 1700);
```


Ceci est faux

```
let enseignant: Enseignant = new Personne(3, "green"
   , "jonas");
```

TypeScript

Remarque

Pour connaître la classe d'un objet, on peut utiliser le mot-clé `instanceof`

TypeScript

Remarque

Pour connaître la classe d'un objet, on peut utiliser le mot-clé `instanceof`

Exemple

```
let enseignant: Personne = new Enseignant(3, "green", "jonas",
   1700);

console.log(enseignant instanceof Enseignant);
// affiche true

console.log(enseignant instanceof Personne);
// affiche true

console.log(personne instanceof Enseignant);
// affiche false
```

TypeScript

Exercice

1. Créer un objet de type `Etudiant`, un deuxième de type `Enseignant` et un dernier de type `Personne` stocker les tous dans un seul tableau.
2. Parcourir le tableau et afficher pour chacun soit le `numero` s'il est personne, soit le `salaire` s'il est enseignant ou soit le `niveau` s'il est étudiant.

TypeScript

Exercice

1. Créer un objet de type `Etudiant`, un deuxième de type `Enseignant` et un dernier de type `Personne` stocker les tous dans un seul tableau.
2. Parcourir le tableau et afficher pour chacun soit le `numero` s'il est personne, soit le `salaire` s'il est enseignant ou soit le `niveau` s'il est étudiant.

Pour parcourir un tableau, on peut faire

```
let personnes: Array<Personne> = [personne, enseignant,
   etudiant];
for(let p of personnes) {

}
```

TypeScript

Solution

```
let personnes: Array<Personne> = [personne,
   enseignant, etudiant];
for(let p of personnes) {

    if(p instanceof Enseignant)
        console.log(p.salaire);
    else if (p instanceof Etudiant)
        console.log(p.niveau)
    else
        console.log(p.num);
}
```

TypeScript

Classe abstraite

- C'est une classe qu'on ne peut instancier
- On la déclare avec le mot-clé `abstract`

TypeScript

Classe abstraite

- C'est une classe qu'on ne peut instancier
- On la déclare avec le mot-clé `abstract`

Si on déclare la classe `Personne` abstraite

```
export abstract class Personne {
    ...
}
```

TypeScript

Classe abstraite

- C'est une classe qu'on ne peut instancier
- On la déclare avec le mot-clé `abstract`

Si on déclare la classe `Personne` abstraite

```
export abstract class Personne {
    ...
}
```

Tout ce code sera souligné en rouge

```
let personne: Personne = new Personne();
...
let personne2: Personne = new Personne(2, 'bob', 'mike');
```

TypeScript

Méthode abstraite

- C'est une méthode non implémentée (sans code)
- Une méthode abstraite doit être déclarée dans une classe abstraite
- Une méthode abstraite doit être implémentée par les classes filles de la classe abstraite

TypeScript

Méthode abstraite

- C'est une méthode non implémentée (sans code)
- Une méthode abstraite doit être déclarée dans une classe abstraite
- Une méthode abstraite doit être implémentée par les classes filles de la classe abstraite

Déclarons une méthode abstraite `afficherDetails()` **dans** `Personne`

```
abstract afficherDetails(): void ;
```

TypeScript

Méthode abstraite

- C'est une méthode non implémentée (sans code)
- Une méthode abstraite doit être déclarée dans une classe abstraite
- Une méthode abstraite doit être implémentée par les classes filles de la classe abstraite

Déclarons une méthode abstraite `afficherDetails()` dans `Personne`

```
abstract afficherDetails(): void ;
```


Remarque

- La méthode `afficherDetails()` dans `Personne` est soulignée en rouge car la classe doit être déclarée abstraite
- En déclarant la classe `Personne` abstraite, les deux classes `Etudiant` et `Enseignant` sont soulignées en rouge car elles doivent implémenter les méthodes abstraites de `Personne`

Pour implémenter la méthode abstraite

- Placer le curseur sur le nom de la classe
- Dans le menu afficher, cliquer sur `Quick Fix` puis `Add inherited abstract class`

Pour implémenter la méthode abstraite

- Placer le curseur sur le nom de la classe
- Dans le menu afficher, cliquer sur `Quick Fix` puis `Add inherited abstract class`

Le code généré

```
afficherDetails(): void {
  throw new Error("Method not implemented.");
}
```

Pour implémenter la méthode abstraite

- Placer le curseur sur le nom de la classe
- Dans le menu afficher, cliquer sur `Quick Fix` puis `Add inherited abstract class`

Le code généré

```
afficherDetails(): void {
  throw new Error("Method not implemented.");
}
```

Remplaçons le code généré dans `Etudiant` par

```
afficherDetails(): void {
  console.log(this.nom + " " + this.prenom + " " + this.niveau);
}
```

Pour implémenter la méthode abstraite

- Placer le curseur sur le nom de la classe
- Dans le menu afficher, cliquer sur `Quick Fix` puis `Add inherited abstract class`

Le code généré

```
afficherDetails(): void {
  throw new Error("Method not implemented.");
}
```

Remplaçons le code généré dans `Etudiant` par

```
afficherDetails(): void {
  console.log(this.nom + " " + this.prenom + " " + this.niveau);
}
```

Et dans `Enseignant` par

```
afficherDetails(): void {
   console.log(this.nom + " " + this.prenom + " " + this.salaire);
}
```

TypeScript

Pour tester

```
let enseignant: Enseignant = new Enseignant(3, "
   green", "jonas", 1700);
enseignant.afficherDetails();
```

TypeScript

Pour tester

```
let enseignant: Enseignant = new Enseignant(3, "
   green", "jonas", 1700);
enseignant.afficherDetails();
```


En exécutant, le résultat est :

```
green jonas 1700
```

TypeScript

En TypeScript

- Une classe ne peut hériter que d'une seule classe
- Mais elle peut hériter de plusieurs interfaces

TypeScript

En TypeScript

- Une classe ne peut hériter que d'une seule classe
- Mais elle peut hériter de plusieurs interfaces

Une interface

- déclarée avec le mot-clé `interface`
- comme une classe complètement abstraite (impossible de l'instancier) dont : toutes les méthodes sont abstraites
- un protocole, un contrat : toute classe qui hérite d'une interface doit implémenter toutes ses méthodes

TypeScript

Définissons l'interface `IMiseEnForme` **dans** `i-mise-en-forme.ts`

```
export interface IMiseEnForme {
    afficherNomMajuscule(): void;
    afficherPrenomMajuscule() : void;
}
```

TypeScript

Définissons l'interface `IMiseEnForme` **dans** `i-mise-en-forme.ts`

```
export interface IMiseEnForme {
    afficherNomMajuscule(): void;
    afficherPrenomMajuscule() : void;
}
```

Pour hériter d'une interface, on utilise le mot-clé `implements`

```
export abstract class Personne implements IMiseEnForme {
    ...
}
```

TypeScript

La classe `Personne` est soulignée en rouge

- Placer le curseur sur la classe `Personne`
- Dans le menu afficher, cliquer sur `Quick Fix` puis `Add inherited abstract class`

TypeScript

La classe `Personne` est soulignée en rouge

- Placer le curseur sur la classe `Personne`
- Dans le menu afficher, cliquer sur `Quick Fix` puis `Add inherited abstract class`

Le code généré

```
afficherNomMajuscule(): void {
    throw new Error("Method not implemented.");
}
afficherPrenomMajuscule(): void {
    throw new Error("Method not implemented.");
}
```

TypeScript

Modifions le code de deux méthodes générées

```
afficherNomMajuscule(): void {
    console.log(this.nom.toUpperCase());
}

afficherPrenomMajuscule(): void {
    console.log(this.prenom.toUpperCase());
}
```

TypeScript

Pour tester

```
let enseignant: Enseignant = new Enseignant(3, "
   green", "jonas", 1700);
enseignant.afficherNomMajuscule();
enseignant.afficherPrenomMajuscule();
```

TypeScript

Pour tester

```
let enseignant: Enseignant = new Enseignant(3, "
   green", "jonas", 1700);
enseignant.afficherNomMajuscule();
enseignant.afficherPrenomMajuscule();
```


En exécutant, le résultat est :

```
GREEN
JONAS
```

TypeScript

Remarque

- Une interface peut hériter de plusieurs autres interfaces (mais pas d'une classe)
- Pour cela, il faut utiliser le mot-clé `extends` et pas `implements` car une interface n'implémente jamais de méthodes.

TypeScript

Une deuxième utilisation

- En TypeScript, une interface peut être utilisée comme une classe **Model** de plusieurs autres interfaces (mais pas d'une classe)
- Elle contient des attributs (qui sont par définition publiques) et des méthodes (abstraites)

TypeScript

Une deuxième utilisation

- En TypeScript, une interface peut être utilisée comme une classe **Model** de plusieurs autres interfaces (mais pas d'une classe)
- Elle contient des attributs (qui sont par définition publiques) et des méthodes (abstraites)

Exemple

```
export interface Person {
    num: number;
    nom: string;
    prenom: string;
}
```

TypeScript

Impossible d'instancier cette interface avec l'opérateur `new`, mais on peut utiliser les objets JavaScript

```
let person: Person = {
    num: 1000,
    nom: 'turing',
    prenom: 'alan'
};
console.log(person);
// affiche { num: 1000, nom: 'turing', prenom: 'alan' }
```

TypeScript

Impossible d'instancier cette interface avec l'opérateur `new`, mais on peut utiliser les objets JavaScript

```
let person: Person = {
    num: 1000,
    nom: 'turing',
    prenom: 'alan'
};
console.log(person);
// affiche { num: 1000, nom: 'turing', prenom: 'alan' }
```


On peut rendre les attributs optionnels

```
export interface Person {
    num?: number;
    nom?: string;
    prenom?: string;
}
```

TypeScript

Ainsi on peut faire

```
let person: Person = {
    nom: 'turing',
};
console.log(person)
// affiche { nom: 'turing' }
```

TypeScript

Ainsi on peut faire

```
let person: Person = {
    nom: 'turing',
};
console.log(person)
// affiche { nom: 'turing' }
```


Pour la suite, gardons l'attribut `nom` obligatoire

```
export interface Person {
    num?: number;
    nom: string;
    prenom?: string;
}
```

TypeScript

Duck typing

- Un concept un peu proche du polymorphisme
- Il se base sur une série d'attributs et de méthodes attendus.
- L'objet est considéré valide quel que soit sa classe s'il respecte les attributs et les méthodes attendus.

TypeScript

Duck typing

- Un concept un peu proche du polymorphisme
- Il se base sur une série d'attributs et de méthodes attendus.
- L'objet est considéré valide quel que soit sa classe s'il respecte les attributs et les méthodes attendus.

Exemple : considérons la fonction `afficherNom()` définie dans `file.ts`

```
function afficherNom(p: Person){
    console.log(p.nom)
}
```

TypeScript

Si l'objet passé en paramètre contient un attribut nom, alors ce dernier sera affiché

```
afficherNom(person);
// affiche turing

afficherNom(personne);
// affiche wick
```

TypeScript

Si l'objet passé en paramètre contient un attribut nom, alors ce dernier sera affiché

```
afficherNom(person);
// affiche turing

afficherNom(personne);
// affiche wick
```

Ceci est aussi correcte car `alien` **a un attribut** `nom`

```
let alien = { couleur: 'blanc', nom: 'white' };
afficherNom(alien);
// affiche white
```

TypeScript

Si l'objet passé en paramètre contient un attribut nom, alors ce dernier sera affiché

```
afficherNom(person);
// affiche turing

afficherNom(personne);
// affiche wick
```

Ceci est aussi correcte car `alien` **a un attribut** `nom`

```
let alien = { couleur: 'blanc', nom: 'white' };
afficherNom(alien);
// affiche white
```

Ceci génère une erreur car `voiture` **n'a pas d'attribut** `nom`

```
let voiture = { marque: 'ford', modele: 'fiesta', num: 100000};
afficherNom(voiture);
```

TypeScript

Décorateur

- L'équivalent d'annotations en **Java** et **php**
- Méta-programmation (modification des informations/comportement sur un objet/classe)
- Utilisé avec le préfixe @

TypeScript

Décorateur

- L'équivalent d'annotations en **Java** et **php**
- Méta-programmation (modification des informations/comportement sur un objet/classe)
- Utilisé avec le préfixe `@`

Ajoutons le décorateur `@f()` à `afficherDetails()` dans `Enseignant`

```
@f()
afficherDetails(): void {
  console.log(this.nom + " " + this.prenom + " " + this.
     salaire);
}
```

`@f()` n'existe pas en TypeScript, il faut donc le définir (en lui associant une fonction)

```
export function f() {
    return function (target: any, propertyKey: string, descriptor:
      PropertyDescriptor){
        console.log('before afficherDetails()');
    }
}
```

`@f()` n'existe pas en TypeScript, il faut donc le définir (en lui associant une fonction)

```
export function f() {
    return function (target: any, propertyKey: string, descriptor:
      PropertyDescriptor){
        console.log('before afficherDetails()');
    }
}
```

Testons maintenant le code suivant et vérifions que la fonction associée au décorateur a bien été exécutée

```
let enseignant: Enseignant = new Enseignant(3, "green", "jonas", 1700);
enseignant.afficherDetails();
// affiche before afficherDetails()
// green jonas 1700
```

`@f()` n'existe pas en TypeScript, il faut donc le définir (en lui associant une fonction)

```
export function f() {
    return function (target: any, propertyKey: string, descriptor:
      PropertyDescriptor){
        console.log('before afficherDetails()');
    }
}
```

Testons maintenant le code suivant et vérifions que la fonction associée au décorateur a bien été exécutée

```
let enseignant: Enseignant = new Enseignant(3, "green", "jonas", 1700);
enseignant.afficherDetails();
// affiche before afficherDetails()
// green jonas 1700
```

Avant de compiler, vérifiez dans `tsconfig.json` les propriétés suivantes

- `"experimentalDecorators": true`
- `"emitDecoratorMetadata": true`

TypeScript

Généricité

- Un concept défini dans tous les LOO avec $<\ \ldots\ >$
- Elle permet de définir des fonctions, classes, interfaces qui s'adaptent avec plusieurs types

TypeScript

Exemple

- si on a besoin d'une classe dont les méthodes effectuent les mêmes opérations quel que soit le type d'attributs
  - **somme** pour **entiers** ou **réels**,
  - **concaténation** pour **chaînes de caractères**,
  - **ou logique** pour **booléens**...
  - ...
- **Impossible sans définir plusieurs classes (une pour chaque type)**

TypeScript

Solution avec la généricité

```
export class Operation<T> {

  constructor(private var1: T, private var2: T) { }

  public plus() {
    if (typeof this.var1 == 'string') {
      return this.var1 + this.var2;
    }
    else if (typeof this.var1 == 'number' && typeof this.var2 == '
       number') {
      return this.var1 + this.var2;
    }
    else if (typeof this.var1 == 'boolean' && typeof this.var2 == '
       boolean') {
      return this.var1 || this.var2;
    }
    else {
      throw "error"
    }
  }
}
```

TypeScript

Nous pouvons donc utiliser la même méthode qui fait la même chose pour des types différents

```
let operation1: Operation<number>  = new Operation(5, 3);
console.log(operation1.plus());
// affiche 8

let operation2: Operation<string> = new Operation("bon", "jour");
console.log(operation2.plus());
// affiche bonjour

let operation3: Operation<number> = new Operation(5.2, 3.8);
console.log(operation3.plus());
// affiche 9

let operation4: Operation<boolean> = new Operation(true, false);
console.log(operation4.plus());
// affiche true
```